Laboratorul 5

- Responsabili laborator: Carmen Odubăsteanu, Mihai Nan
- Profesor titular: Carmen Odubăsteanu

Exceptii

Introducere

Un program trebuie sa tina cont de posibilitatea aparitiei, la executie, a unor situatii limita neobisnuite. Spre exemplu, daca avem un program care incearca sa deschida un fisier, care nu exista pe disc, pentru a citi din el. Ideal, toate situatiile neobisnuite, ce pot sa apara pe parcursul executiei unui program, trebuie detectate si tratate corespunzator de acesta.

In general, o eroare este o exceptie, dar o exceptie nu reprezinta neaparat o eroare de programare. Astfel, o exceptie reprezinta un eveniment deosebit ce poate sa apara pe parcursul executiei unui program si care necesita o deviere de la cursul normal de executie al programului.

Metode de tratarea exceptiilor

Dupa cum am vazut deja pana acum, in limbajul Java, aproape orice este vazut ca un obiect. Prin urmare, este destul de intuitiv faptul ca o exceptie nu este altceva decat un obiect care se defineste aproape la fel ca orice alt obiect. Cu toate acestea, exista si o conditie suplimentara ce trebuie satisfacuta: clasa care defineste obiectul trebuie sa mosteneasca, direct sau indirect, clasa predefinita [http://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html [http://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html] **Throwable**].

In practica, la definirea exceptiilor, **NU** se extinde direct aceasta clasa, ci se utilizeaza clasa **Exception**, o subclasa a clasei **Throwable**.

Pentru a intelege mai bine notiunea de exceptie, vom considera un exemplu ipotetic simplu. Sa presupunem ca avem un vector intrinsec cu elemente de tip **int**. Daca ii alocam o dimensiune fixa, in timpul instantierii, nu vom mai putea introduce elemente in acest vector atunci cand atingem dimensiunea maxima. De asemenea, nu putem incerca sa eliminam un element din vector daca acesta este gol. In continuare, se vor prezenta doua implementari ce se pot uzita intr-un astfel de scenariu.

```
class ExceptieSirPlin extends Exception {
    public ExceptieSirPlin() {
        super("Sirul a atins capacitatea maxima!");
    }
}

class ExceptieSirVid extends Exception {
    public ExceptieSirVid(String text) {
        super(text);
    }
}
```

Clauza throws

Inca din primele laboratoare am spus ca obiecte interactioneaza prin apeluri de metoda. Din perspectiva unui obiect care apeleaza o metoda a unui alt obiect, la revenirea din respectivul apel putem fi in doua situatii: metoda s-a executat in mod obisnuit sau metoda s-a terminat in mod neobisnuit, datorita aparitiei unei exceptii. Clauza **throws** apare in antetul unui metode si ne spune ce tipuri de exceptii pot conduce la **terminarea neobisnuita** a respectivei metode. Mai jos, vom prezenta modul in care specificam metodele clasei **SirNumere** care pot sa termine executia datorita unei situatii neobisnuite.

```
class SirNumere {
    Vector vector;
```

```
    //Implementarea propriu-zisa nu ne intereseaza (momentan)
    public void adauga(int x) throws ExceptieSirPlin {

    }

    //Implementarea propriu-zisa nu ne intereseaza (momentan)
    public int sterge(int x) throws ExceptieSirVid {
        //returneaza pozitia elementului sters
    }
}
```

Clauza **throws** poate fi vazuta ca o specificare suplimentara a tipului returnat de o metoda. De exemplu, metoda **sterge** returneaza, in mod obisnuit, o valoare de tip **int**, reprezentand pozitia elementului sters. Clauza **throws** spune ca, in situatii exceptionale, metoda poate returna o referinta la un obiect de tip **ExceptieSirVid** sau o referinta la un obiect al carui tip reprezinta un subtip al acestei clase.

Este important de subliniat faptul ca dupa clauza **throws** se pot introduce mai multe nume de exceptii, separate prin virgula.

Tratarea exceptiilor

Pana acum am vorbit despre modul de definire a unei exceptii si de modul in care specificam faptul ca o metoda poate sa se termine cu una sau mai multe exceptii. In continuare, ne vom ocupa de raspunsul urmatoarei intrebari: ,,Cum poate afla un obiect, folosit pentru apelul unei metode ce se poate termina cu una sau mai multe exceptii, daca apelul s-a terminat normal sau nu?''. Raspunsul este unui simplu, deoarece, evident, cei care au proiectat Java s-au gandit la o rezolvare pentru o astfel de situatie. Acest lucru se rezolva uzitand un bloc **try-catch-finaly**, structura generala pentru un astfel de bloc fiind prezentata mai jos.

```
try {
        //Secventa de instructiuni in care ar putea sa
        //apara exceptii
} chatch(TipExceptie1 e) {
        //Secventa de instructiuni ce se executa cand, in
        //sectiunea try apare o exceptie, avand tipul sau
        //subtipul TipExceptie1 (parametrul fiind o referinta
        //la exceptia prinsa)
} catch(TipExceptie2 e) {
        //Acelasi lucru doar ca este vorba de TipExceptie2
}
//Sectiunea catch poate lipsi din acest bloc
//................................................
catch (TipExceptiN e) {
        //Acelasi lucru doar ca este vorba de TipExceptien
} finaly {
        //Secventa de instructiuni ce se executa in orice
        //conditii la terminarea executiei celorlalte sectiuni
        //Aceasta este obtionala (poate exista maxim una)
}
//Alte instructiuni ale metodei
```

In sectiunea **try** se introduce codul pe parcursul caruia pot sa apara exceptii. In fiecare sectiune **catch** se amplaseaza secventa de intructiuni ce trebuie sa se execute in momentul in care sectiunea in **try** a aparut o exceptie de tipul parametrului sectiunii **catch** (sau un subtip). In sectiunea **finally**, se amplaseaza cod ce trebuie neaparat sa se execute inaintea parasirii blocului **try-catch-finally**, indiferent daca a aparut sau nu vreo exceptie (tratata sau netratata).

Emiterea explicita a exceptiilor

In aceasta sectiune vom vedea cum anume se anunta explicit aparitia unei situatii neobisnuite. Mai exact, vom vedea cum se emite explicit o exceptie. Acest lucru se va realiza uzitand instructiunea **throw**.

throw ExpresieDeTipExceptie;

In continuare, vom prezenta implementarea integrala a clasei **SirNumere**, pentru o intelegere mai buna a conceptelor explicate in aceasta sectiune.

```
class SirNumere {
    int vector[], dim, poz = 0;
    public SirNumere(int dim) {
        vector = new int[dim];
        this.dim = dim;
    }
    public void adauga(int x) throws ExceptieSirPlin {
        if(vector.length == dim) {
            throw new ExceptieSirPlin();
        } else {
            vector[this.poz++] = x;
        }
    }
    public int sterge(int x) throws Exception {
        if(this.poz == 0) {
            throw new ExceptieSirVid("Sirul este vid");
        } else {
            int pozitie = -1;
            for(int i = 0; i < this.poz && pozitie==-1; i++) {
                if(vector[i] == x) {
                    pozitie = i;
                }
            }
            if(pozitie != -1) {
                for(int i = pozitie; i < this.poz - 1; i++)
                    vector[i] = vector[i+1];
                return pozitie;
            } else {
                throw new Exception("Valoarea nu exista!");
            }
        }
    }
    public static void main(String args[]) {
        SirNumere sir = new SirNumere(10);
        try {
            for(int i = 0; i < 10; i++) {
                int nr = sir.sterge(sir.vector[i]);
            }
            while(true) {
                sir.adauga((int) Math.random() * 100);
            }
        } catch(ExceptieSirPlin e) {
            e.printStackTrace();
        } catch(ExceptieSirVid e) {
            e.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
        } finally {
            System.out.println("Am terminat");
        }
    }
}
```

Exceptii implicite

In Java, exista instructiuni care pot emite exceptii intr-o maniera implicita, in sensul ca nu se utilizeaza instructiunea throw pentru emiterea lor. Aceste exceptii sunt emise de masina virtuala Java, in momentul detectiei unei situatii anormale la executie.

Spre exemplu, vom considera un vector intrinsec, avand elemente de tip double si o capacitate de 10 elemente. In momentul in care vom incerca sa accesam un element folosind un index mai mare ca 9 sau mai mic ca 0, masina virtuala Java va emite o exceptie de tip **IndexOutOfBoundsException**.

```
class Test {
    double v[];

    public Test(int dim) {
        v = new double[dim];
    }

    public static void main(String args[]) {
```

```
        Test t = new Test(10);
        for(int i = 0; i < 9; i++) {
            t.v[i] = (double) Math.random()*100;
        }
        System.out.println(t.v[10]);
    }
}
```

Blocuri de initializare

Pentru a face prelucrari si pentru a obtine rezultate este nevoie de date. Si aceste valori de intrare sunt de obicei stocate in **variabile statice**, **variabilele locale** (definite in metode) sau **variabile de instanta** (variabile nonstatice definite in clase). Pentru a initializa o variabila o poti face la definitia sau mai tarziu intr-o metoda (constructor sau nu). In ciuda acestor doua solutii comune, exista o alta cale folosind blocuri de initializare.

Ca majoritatea lucrurilor in Java, blocuri de initializare sunt de asemenea definite intr-o clasa si nu la nivel global.

In continuare, se va prezenta un prim exemplu minimal de uzitare a blocurilor de initializare.

```
class Test {
    //bloc static de initializare
    static {
        System.out.println("Bloc static de initializare!");
    }

    //bloc de initializare
    {
        System.out.println("Bloc de initializare!");
    }

    public static void main(String[] args) {
        System.out.println("Aceasta este metoda main()!");
    }
}
```

Dupa cum am vazut in exemplul anterior, exista doua tipuri de blocuri de initializare cu comportamente diferite.

Blocuri statice de initializare

Blocurile statice de initilizare sunt blocuri de cod care sunt executate **DOAR O DATA** atunci cand clasa este incarcata de Java Virtual Machine; acesta este motivul pentru care mesajul din blocul static de initializare este imprimat inainte de apelul metodei **main**. Aceste tipuri de blocuri de initializare sunt utile pentru initializarea atributelor statice din clase sau pentru a efectua o singura data un set de prelucrari.

Bocurile statice de initializare pot accesa **DOAR** atributele statice ale clasei in care sunt definite. **NU** se pot folosi variabile de instanta in blocuri de initializare statice (daca incercati sa faceti acest lucru, veti primi o eroare de compilare: **non-static variable value cannot be referenced from a static context**).

Blocuri instanta de initializare

Blocurile instanta de initializare sunt blocuri de cod care sunt executate de fiecare data cand o instanta a clasei este creata (in cazul in care constructorul este apelat). Aceste blocuri de initializare urmaresc initializari de atribute statice sau prelucrari generice ce trebuie executate de fiecare data cand se construieste un obiect indiferent de constructorul apelat.

Blocurile instanta de initializare pot accesa atribute statice si variabile de instanta (variabilele de instanta reprezinta atributele instantei construite) deoarece acestea sunt executate chiar inainte de construirea instantei.

Blocurile statice de initializare sunt executate in secventa:

1. blocuri statice din clasa de baza;
2. blocuri statice din clasa derivata.

Aceasta secventa se respecta, deoarece fiecare bloc static de initializare este executat dupa ce clasa in care este definit este incarcata. Blocuri de initializare sunt executate dupa apelul constructorului din clasa de baza – **super()**.

poo/breviare/breviar-05.txt · Last modified: 2018/09/23 12:36 by mihai.nan